Optimising Skeletal-Stream Parallelism on a BSP Computer

Andrea Zavanella[*]

Dipartimento di Informatica
Università di Pisa Italy
`zavanell@di.unipi.it`
`http://www.di.unipi.it/~zavanell`

Abstract. Stream parallelism allows parallel programs to exploit the potential of executing different parts of the computation on distinct input data items. Stream parallelism can also exploit the concurrent evaluation of the same function on different input items. These techniques are usually named "pipelining" and "farming out". The P^3L language includes two stream parallel skeletons: the `Pipe` and the `Farm` constructors. The paper presents a methodology for efficient implementation of the P^3L `Pipe` and `Farm` on a BSP computer. The methodology provides a set of analytical models to predict the constructors performance using the BSP cost model. Therefore a set of optimisation rules to decide the optimal degree of parallelism and the optimal size for input tasks (grain) are derived. A prototype has been validated on a Cluster of PC and on a Cray T3D computer.

1 Introduction

Parallel computers can exploit an input stream to execute in parallel different parts of code while data items flow through the PEs. Generally the optimisation of a pipeline computation requires the programmer to deal with the bottleneck problem (i.e. a stage is particularly slower than the others). The goal of such a decomposition is to minimise the service time of the module, which is now limited by the maximum of the stages service times, while the latency per item is normally greater than in sequential monolithic versions. A different solution to exploit the input stream arises when a function f can be computed, over a single item, independently from the evaluation of the others. In this case computation can be "farmed out" to a group of PEs (generally named "workers"). The goal of farming the computation of the function is mainly to minimise the service time of the module. Implementing a "farm" strategy charges programmers of decisions as the degree of parallelism (i.e to decide the number of workers), the scheduling policy to adopt and the size of the "task" to schedule. In this article we propose a strategy to efficiently implement the two decomposition schemes presented

[*] This Work has been partially supported by the italian M.U.R.S.T. in the framework of the Project MOSAICO and by the UE with a TMR (Marie Curie) grant

above on a generic BSP computer [8, 6].The strategy is proposed with the goal in mind of designing P^3L as a portable skeleton language [3, 1]. In particular the paper provides a set of optimisation rules for a portable implementation of the P^3L constructors `Pipe` and `Farm`.

2 P^3L and the BSP Computer

A suitable approach to reconcile performance and abstraction in parallel programming is provided by the skeletons methodology [2, 4, 7]. In skeletal programming the parallelism is introduced using a limited set of parallel patterns which are automatically implemented by the support. The P^3L language allows several parallel constructors to be composed and nested in order to write fully structured parallel programs. The parallel constructors included by the language are: `Pipe, Farm, Loop, Map, Reduce, Scan` and `Comp`.

The BSP computer is a parallel abstract machine [8] composed by: *a set of memory-processor couples, an interconnection structure*, and a *synchronisation device* A BSP computation is a sequence of so-called *supersteps.* Each superstep includes three phases: *local computation phase*, a *interprocessor communication phase* and a *global synchronisation phase.* In the BSP model the differences between parallel machines are reflected by a small set of *machine-dependent parameters*: p is the number of available processors; g_∞ is the asymptotic communication cost for very large messages expressed in time steps; $N_{1/2}$ is the size of the message that produces half the optimal bandwidth, in other words $g(N_{1/2}) = 2g_\infty$ and a message of size h can be routed in $g(h) * h$ time steps; l number of time steps for barrier synchronisation; s is the number of time steps per second. Assuming that W is the maximum number of operations executed by one processor during the local computation phase and h is the maximum amount of data moved by one processor in the interprocessor communication phase: the cost of a superstep is given by $T_{sstep} = W + g_\infty h(1 + \frac{N_{1/2}}{h}) + l$. The issues for efficient implementations of the BSP model have been addressed on several papers [6] and the framework has already been used to derive optimisation rules for data parallel programs and data parallel skeletons [5, 9].

3 Implementing P^3L Pipe on the BSP Computer

A cost-efficient implementation for the `Pipe` constructor is derived using the following variables: the inter-arrival time (per item) is T_e; a set of functions $f_i : \mathcal{T}_i \longrightarrow \mathcal{T}_{i+1}$; the size of T_i is d_i; the time to compute f_i is t_i A simple scheme to implement a pipeline on a BSP computer exploits a double buffering technique which implies that during each superstep each PE performs three operations: a) computing the function on the previously received item; b) receiving the next input item; c) sending the result to the next stage. The service time of `Pipe` using tasks of k items is $T^{pipe}_{serv}(k) = Max(kT_e, T^{stag}_{serv}(k))$, where:

$$T^{stag}_{serv}(k) = Max_{[0<i<z-1]} T^i_{serv}(k) \tag{1}$$

$$T^{i}_{serv}(k) = (t_i k + l + g_\infty k(d_i + d_{i+1}) + 2g_\infty N_{1/2}) \tag{2}$$

Since balance between the external time and service times of the stages is convenient we propose two optimisation strategies:

– *adjusting the grain*: the value k of the number of items per task can be tuned. Assuming that $\mathcal{T} = Max_{[0<i<z-1]}(t_i + g_\infty(d_i + d_{i+1})$ we consider the two cases: $\mathcal{T} < T_e$ and $\mathcal{T} \geq T_e$. In the first case the value of k balancing the service time of the `Pipe` with the interarrival time is given by: $k = \frac{l+2g_\infty N_{1/2}}{T_e - \mathcal{T}}$ In the second case the value $k = 1$ makes the service time of the `Pipe` minimal.
– *merging fast stages*: this technique provides an implementation which uses *less resources* of the starting one and which has the same service time. We can apply this technique when $\exists j : S(j) + S(j+1) < Max_{[0<j<z-1]}S(i)$ where: $\mathcal{S}_(i) = f_i + g_\infty(d_i + d_{i+1})$. Under this assumption we can merge the stage j and $j+1$ of the pipe in a sequential stage which does not increase the service time.

4 Implementing P^3L Farm on the BSP Computer

Assuming that the interarrival time of the input stream is T_e and the time to compute f on a single item is t_f. In the case $t_f >> T_e$ and f cannot be further decomposed we can attempt to speed-up our computation using a farming technique. The template proposed uses a synchronous pipeline scheme of three types of processes: emitter, workers (n) and collector. The service times of emitter-collector and workers per item are:

$$T^{em-col}_{serv}(n,k) = 2g_\infty d_0 + \frac{g_\infty N_{1/2}}{k} + \frac{g_\infty N_{1/2} + l}{nk} \tag{3}$$

$$T^{work}_{serv}(n,k) = \frac{2g_\infty d_0 + t_f}{n} + \frac{2g_\infty N_{1/2} + l}{nk} \tag{4}$$

The service time in Eq. 3 introduces two optimisation cases, when: $2g_\infty d_0 \leq T_e$ both the Eq. 3-4 can be decreased to the interarrival time T_e choosing a couple of values: $(\hat{n}, \hat{k})$ such that be $\mathcal{P}$ the property:

$$\mathcal{P}(n,k) \equiv T^{em}_{serv}(n,k) < T_e \,\&\, T^{work}_{serv}(n,k) < T_e \tag{5}$$

Then $P(\hat{n}, \hat{k})$ and: $\forall(n,k) \neq (\hat{n}, \hat{k})\mathcal{P}(n,k) \Rightarrow n > \hat{n}$ From Eq. 3-4 using a practical optimisation rule:

$$\hat{n} = Max(|\frac{g_\infty N_{1/2}}{(T_e - 2g_\infty d_0) + g_\infty N_{1/2}}|, |\frac{2g_\infty d_0 + t_f}{T_e}| + 1) \tag{6}$$

$$\hat{k} = Max(1, |\frac{2g_\infty N_{1/2} + l}{T_e}|) \tag{7}$$

When: $2g_\infty d_0 > T_e$ we use a different rule to compute a value of k giving an approximation of the minimum for Eq 3 such that: $T^{em}_{serv} < (2g_\infty d_0)(1+1/prec)$. The precision parameter *prec* can be tuned as a compiling option.

$$\hat{k} = \frac{prec(g_\infty N_{1/2} + l)}{2g_\infty d_0} \tag{8}$$

Fixed the value of $\hat{k}$ we can derive $\hat{n}$:

$$\hat{n} = \frac{prec(2g_\infty d_0 + t_f + 2g_\infty d_0)}{(prec+1)2(g_\infty d_0)} \approx 1 + |\frac{4g_\infty d_0 + t_f}{2g_\infty d_0}| \tag{9}$$

5 Experiments

The optimisation rules of Section 4 have been tested on two different parallel architectures: a Cluster of 10 PC with PentiumII 266 Mhz processors running Linux connected by a 100Mbit Fast Ethernet technology (Backus), and a Cray T3D with 512 Alpha processors with clock rate of 150 Mhz connected by a three dimensional torus having a 300 Mbyte/s bandwidth per link. A group of tests have been executed using the C+BSP-lib and using a stream of items (C-type: `double`). The predictions of the model prove to be accurate and we see that the values choosen by the model for $\hat{n}$ and $\hat{k}$ are extremely close to the optimal ones.

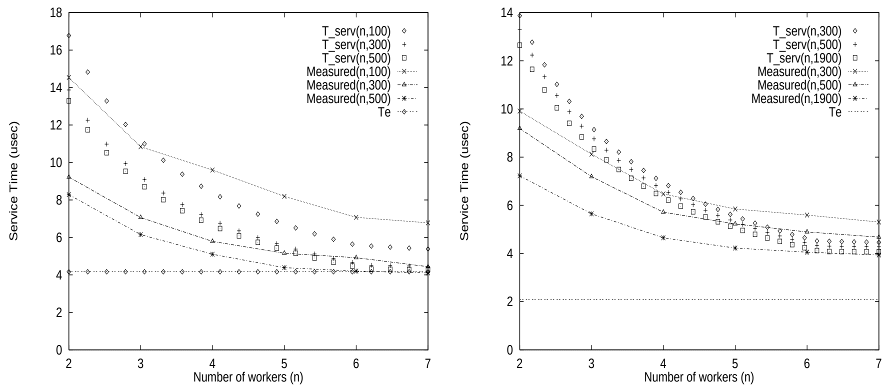

Fig. 1. Predicted and Measured times: Test Backus

6 Conclusions and Related Works

A methodology for an optimised implementation of Stream Parallelism on a BSP computer has been proposed and validated. The model provides accuracy

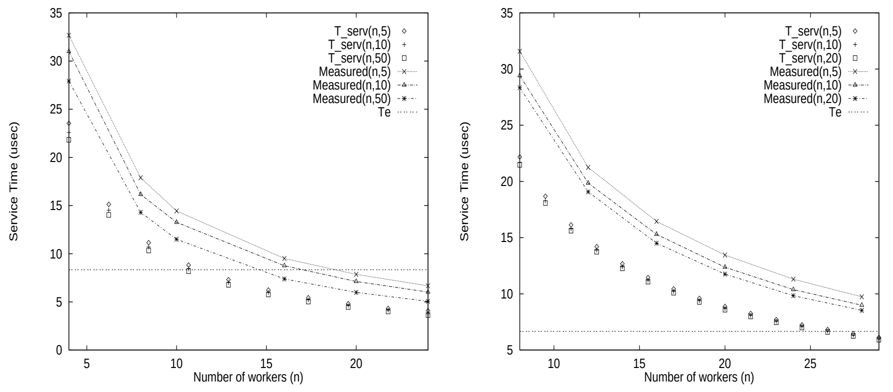

Fig. 2. Predicted and Measured times: Test T3D

enough to automatically decide the degree of parallelism and grain of data. This work also shows how a skeleton languages like the P^3L can be implemented achieving both performance and portability. The methodology together with other recent works [9, 5] is a further step towards the design of a complete compiling technology having as its first goal the performance portability.

References

[1] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a Template-based p3l Compiler. In *Proceedings of the Seventh Parallel Computing Workshop (PCW '97)*, Australian National University, Canberra, August 1997.

[2] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* The MIT Press, Cambridge, Massachusetts, 1989.

[3] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A Methodology for the Development and the Support of Massively Parallel Programs. In D.B. Skillicorn and D. Talia, editors, *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1994.

[4] J. Darlington, M. Ghanem, and H. W. To. Structured Parallel Programming. In *Proceedings of the Working Conference MPPM '93*, Berlin, September 1993. GMD FIRST.

[5] D.B.Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising Data-Parallel Programs Using the BSP Model. In *Proceeding of EUROPAR98*, LNCS. Springer, 1998.

[6] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. Technical Report PRG-TR-15-96, Oxford University Computing Laboratory, 1996.

[7] S.Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, 1997.

[8] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[9] A. Zavanella and S. Pelagatti. Using BSP to Optimize Data-Distribution in Skeleton Programs. In *Proceedings of HPCN99*, number 1593 in LNCS, pages 613–622, April 1999.